CIRCUIT CELLAR®
THE MAGAZINE FOR COMPUTER APPLICATIONS

FEATURE ARTICLE

by Fred Eady

2007 WIZnet iEthernet DESIGN CONTEST

iEthernet Bootcamp

Get Started with the W5100

Are you ready to join the Ethernet revolution? If so, it's time to start working with WIZnet's W5100 hardwired TCP/IP embedded Ethernet controller. In this article, Fred helps you get started on your first W5100-based design.

I recently received an e-mail from a reader asking why there were no in-depth TCP/IP stack "how-to" articles. Honestly, I had never given that much thought because I normally forego the formal TCP/IP stack in favor of small, easy-to-follow, home-brewed Ethernet driver packages. As a magazine writer, my first guess on the lack of TCP/IP stack magazine literature is the cost versus interest factor. I have reviewed many commercial TCP/IP stack products and I can say from experience that you get what you pay for. My readers simply can't afford or financially justify a full-blown commercial TCP/IP stack for their applications and projects. Thus, why should I ask them to read about a TCP/IP product that they can't afford to use? My second stab at why TCP/IP stacks aren't in magazine vogue is complexity. Many of you have written articles for magazines and you know that you are limited to so many words per article. It would take a series of articles to explain everything you would need to know about TCP/IP stacks.

I must admit that in the past I have offered up some pretty pricey stuff in my articles. These days, I tend to shy away from super-expensive and complex subjects for the reasons I just outlined. However, when I see something that may be what typical technical magazine readers like you and I are looking for, I'm all over it. For instance, I have found that Ethernet ICs supported by free TCP/IP stacks are very popular with *Circuit Cellar* readers. I've also discovered that many readers who implement single-IC Ethernet devices don't even use a TCP/IP stack. Instead, like me, they employ simple protocol drivers specifically written for the single-IC Ethernet device that they are deploying in their project. I practice what I preach, and what I'm about to introduce to you is the best of both the garage Ethernet driver and TCP/IP stack worlds. How would you like to solder down a single-IC Ethernet solution that provides the power of a full-blown commercial TCP/IP stack as if it were a set of simple Ethernet drivers? Read on, my friend.

WIZnet W5100

The WIZnet W5100 is a single-IC Ethernet solution with a built-in TCP/IP stack. The W5100 folks like to call their on-chip stack a "hardwired stack" because all of the W5100's Internet-enabling goodies are contained within a compact 80-pin LQFP. In addition to the W5100's hardwired TCP/IP stack, other W5100 Ethernet goodies include an integrated IEEE 802.3 10Base-T and 802.3u 100Base-TX-compliant MAC and PHY. As you would expect, the W5100's TCP/IP stack supports all of the things you need to put an embedded Ethernet gadget on a network. The W5100's TCP/IP stack supports TCP, UDP, ICMP, and ARP, which normally provide enough protocol power for a major portion of embedded Ethernet LAN and Internet projects that are launched by folks like you and me. PPPoE is also supported by the W5100. The inclusion of PPPoE enables you to use the W5100 in ADSL applications.

If you've ever toyed with embedded Ethernet, you know that the lack of a transmit or receive buffer memory can

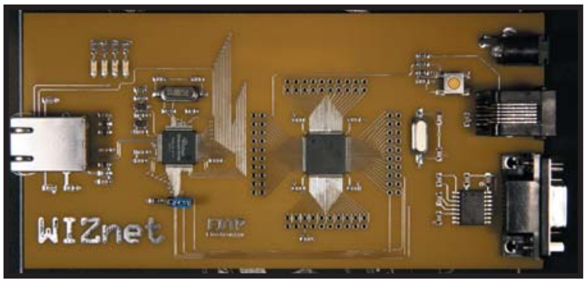


Photo 1—My WIZnet W5100 development board is based on the Microchip Technology PIC18LF8722. The PIC18LF8722 is hefty enough to enable the selective use of Direct Memory mode, Indirect Memory mode, and SPI mode access to the W5100's registers and buffer memory. Using the Microchip PIC18LF8722 also puts the powerful set of Microchip development tools at our disposal.

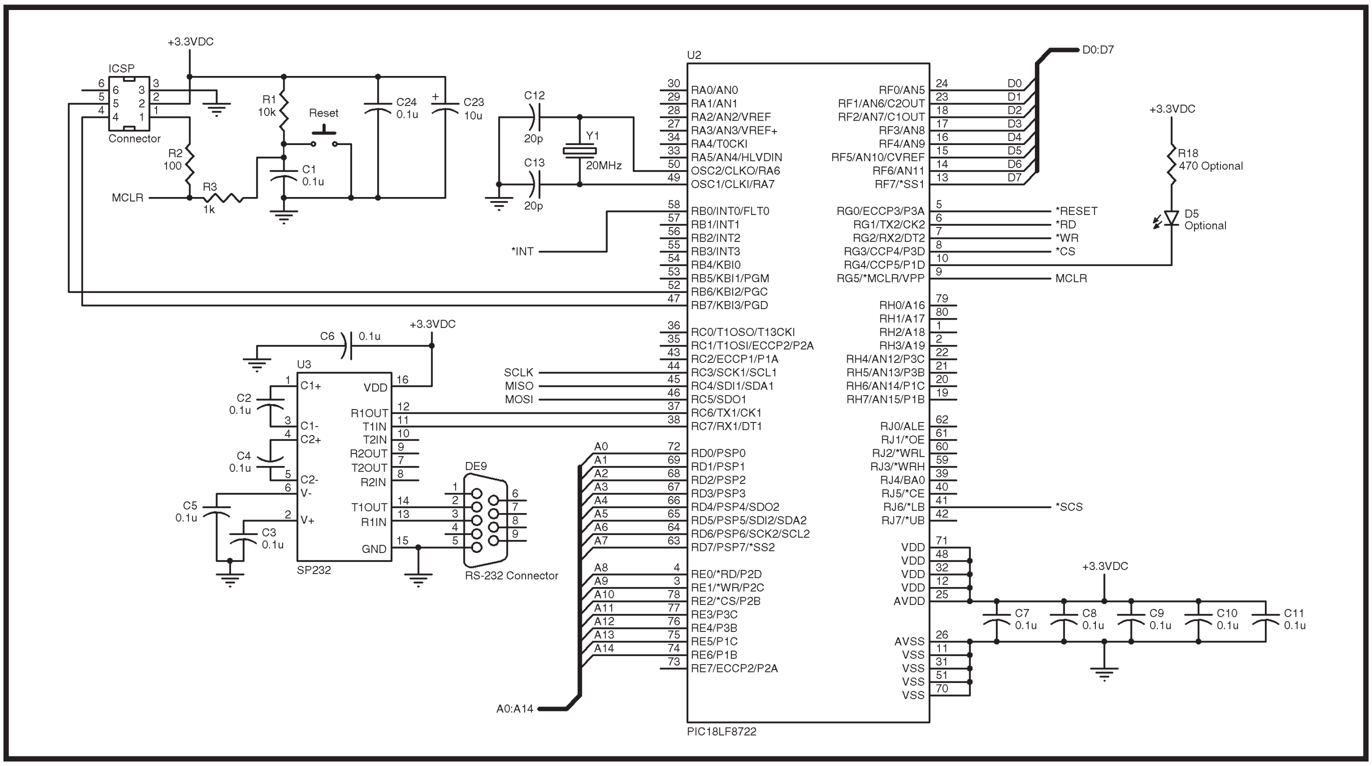


Figure 1—Nothing much I need to say about what you see here. However, the PIC18LF8722 does remind me of my favorite Military Channel quote: "It's just a good, solid tank."

be painful and hamper the performance of your embedded Ethernet device. The embedded Ethernet IC manufacturers are aware of this. Most of the single-IC Ethernet solutions offered these days include a fair amount of dedicated transmit and receive buffer memory. The W5100 is no exception, and it is equipped with 16 KB of internal transmit/receive buffer memory.

To avoid the exclusion of smaller microcontrollers, the W5100 can communicate with a host of microcontrollers using an SPI, direct memory access, or indirect memory access. To further accommodate the majority of today's newer microcontrollers, the W5100 is powered with a 3.3-VDC power source. This enables the W5100 to be directly interfaced to low-power microcontrollers that also run on a 3.3-VDC power rail. The W5100 can also be integrated into legacy 5-VDC systems because its I/O subsystem is 5-V tolerant.

The W5100 supports up to four simultaneously active sockets. Thus, all you need to know is basic socket programming because you will be shielded from the W5100's internal Ethernet engine operations. The W5100 is designed to provide the bulk of everything needed to produce a working embedded Ethernet device while being easy to use. The only things the W5100 won't do for you are write its own code and handle IP fragmentation.

I just happen to have a couple of W5100 ICs. Let's assemble a W5100-based device from scratch. Once we've got the W5100 hardware realized, we'll put together some Microchip Technology PIC18LF8722 driver code for our W5100 development board.

BUILD A DEVELOPMENT BOARD

For your convenience, I am supplying the PCB layout for an EDTP Electronics-designed W5100 device (see Photo 1). The PCB layout file on the *Circuit Cellar* FTP site is in ExpressPCB format. I chose ExpressPCB because it is a relatively inexpensive PCB manufacturing service that is available to everyone. ExpressPCB software is free for download, and the quality of ExpressPCB PCBs is excellent. Another plus associated with using ExpressPCB is that you don't have to design your W5100 PCB from scratch. You can use my ExpressPCB PCB template and modify it to meet your needs. If you already have a favorite PCB CAD program, you can easily port my design to your CAD format using my original drawing as a guide. As you would expect, I haven't done anything to complicate the W5100 project board design.

The EDTP WIZnet W5100 project board is basically a standard PIC18LF8722 configuration that is wired into a basic W5100 configuration. As you can see in Figure 1, the PIC18LF8722 has enough I/O to wire-in the 15-bit W5100 address bus, the 8-bit W5100 data bus, and all of the W5100 control signals (*RD, *WR, *CS, and *INT) with I/O to spare. In addition to wiring in the W5100 in Direct Bus Interface mode (A0:A14 with D0:D7 and control signals), I attached the W5100's SPI portal and an SPI select pin to the PIC18LF8722's SPI I/O interface, which enables you to access the W5100's internals in W5100 SPI mode. Because the W5100's address lines are all pulled down internally, the Indirect Bus Interface mode of operation, which uses only two of the 15 address lines, all of the eight data lines, and all of the control signals can also be easily implemented with the EDTP WIZnet W5100 design.

All of the PIC18LF8722's 80 I/O and power lines are pinned out in blocks of 20 pins to standard 0.1" header pads. The PIC18LF8722 is supported by a 20-MHz clock, a Microchip-certi-

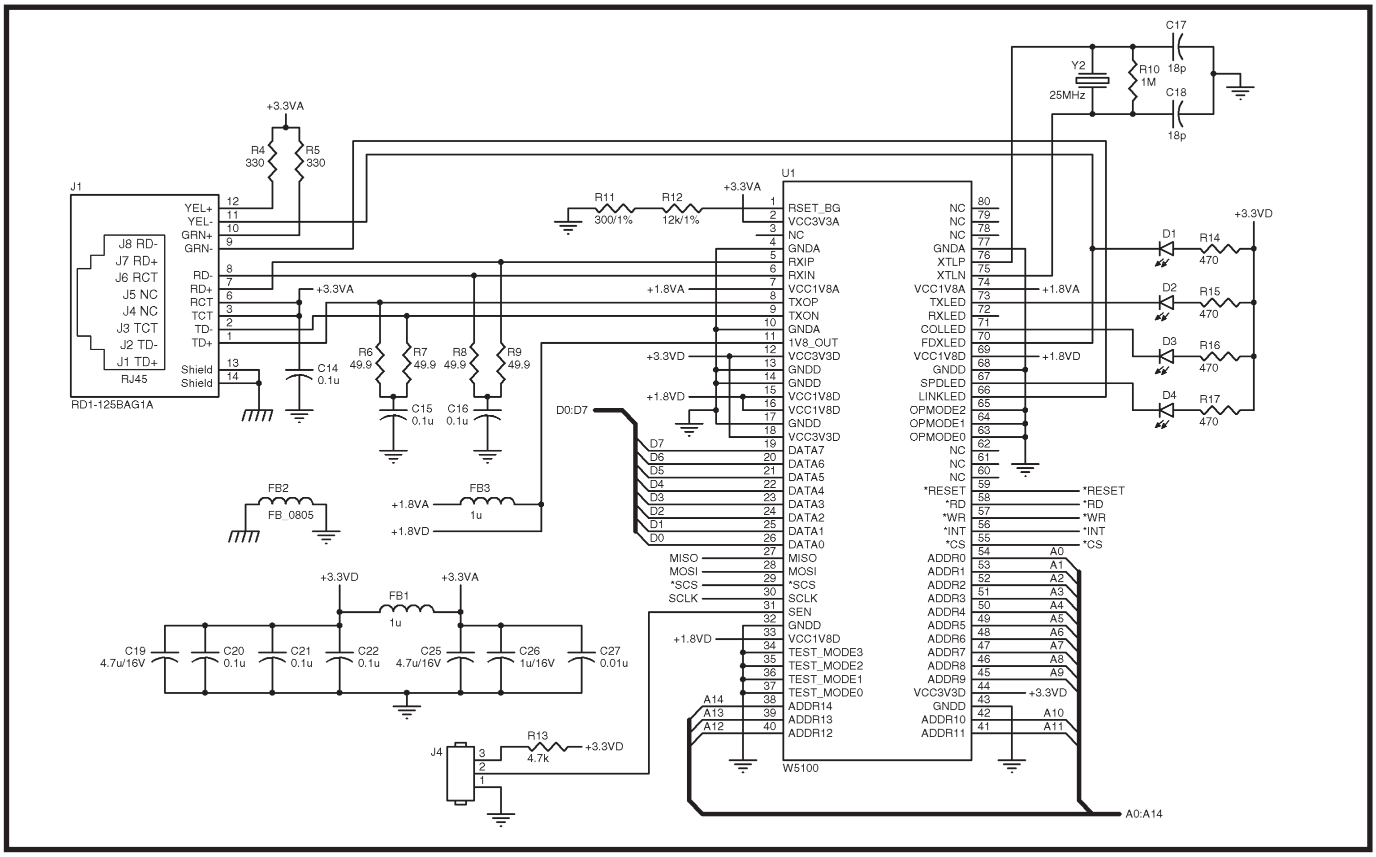


Figure 2—You can get your hands on most everything here from Digi-Key or Mouser Electronics. My friends at Saelig supply the W5100 IC. Saelig doesn't stock the RD1-125BAG1A on its site, but you can probably get the pulse transformer from many of the vendors listed on WIZnet's web site.

fied ICSP programming/debugging portal, and a regulation RS-232 port. I did not include any power supply circuitry because a Digi-Key-supplied 3.3-VDC wall wart does a great job powering the W5100 project board and the external programming/debugging hardware.

On the W5100 side of the EDTP W5100 development board, the W5100 is supported by the required 25-MHz crystal and an all-in-one can of magnetics (see Figure 2). I chose to incorporate the U.D. Electronic RDI-125BAG1A pulse transformer for a couple of reasons. First, the RDI-125BAG1A footprint fits exactly into the old packet whacker pulse transformer footprint, for which I already have a time-proven ExpressPCB pad layout. Second, like the old packet whacker mag jack package, the RDI-125BAG1A has a pair of built-in indicator LEDs in addition to a pair of transmit and receive pulse transformers and the required internal terminating resistors. If you've ever worked with the EDTP ASIX-based and Microchip-based Ethernet development boards, you'll notice that the W5100 PHY connections are very similar to the EDTP Electronics ASIX and Microchip ENC29J60 designs.

You may wonder why there are no bypass components on the W5100's internally generated 1.8-VDC supply. That question was posted on the W5100 online technical support question-and-answer board. The W5100 engineering answer was to follow the path that was set forth by the W5100 reference schematic, which is void of 1.8-VDC supply bypass components.

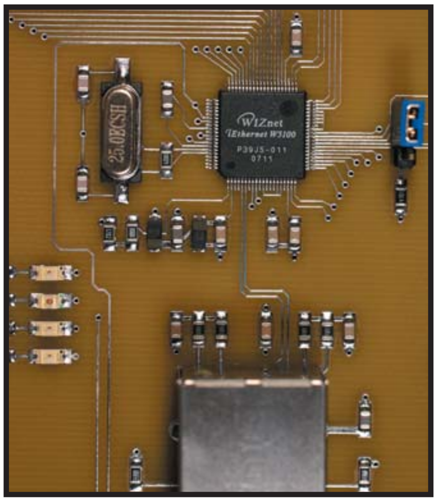


Photo 2—There's nothing here you can't handle. With the exception of the 12.3-kΩ resistor pair, the line of components closest to the W5100 is all filter and bypass components. The PHY components are in the line closest to the pulse transformer. Note the status LEDs and the SPI select jumper at the port and starboard extremes of this photo.

Because the EDTP WIZnet W5100 project board is designed to help you get your W5100 design up and running quickly, I attached all of the W5100 LED indicator lines to LEDs. The pair of RDI-125BAG1A LEDs is connected to the W5100's LINKLED and RXLED status indicator I/O pins. I pulled the TXLED, COLLED, FDXLED, and SPDLED indicators out to discrete LEDs, which you can see in Photo 1 hanging above the city of WIZnet W5100 0805 supporting SMT components. I have also provided a jumper to select W5100 SPI mode if you choose to run your W5100 in that manner. The only oddity I need to point out is the 12.3-kΩ reset resistor pair you see in Figure 2, which is attached to the W5100's RSET_BG pin. A bird's-eye view of the W5100 portion of the EDTP WIZnet W5100 development board is in Photo 2.

As you can see, the W5100 hardware is a no-brainer. Before we move on to do some W5100 coding, what you don't see in Photo 1 is the heartbeat LED I attached to RG4 on the PIC18LF8722. It's just there as a warm fuzzy to let me know that things are moving on the firmware side. I flash the RG4 LED at a rate of 1 Hz via the PIC18LF8722's Timer3 interrupt-driven real-time clock code.

WIZNET W5100 GARAGE CODE

From a WIZnet W5100 programmer's point of view, the W5100 consists of Common registers, Socket registers, TX memory, and RX memory. The W5100's Common registers consist mostly of W5100 local IP and MAC addressing fields. Also included within the confines of the Common registers are RX and TX memory sizes and PPP/PPPoE parameters. It looks like we will be populating most of the Common registers. So, let's kill two birds with one stone and use the Common registers to test the PIC18LF8722 driver hardware by writing some basic PIC18LF8722 routines to read and write the W5100's registers. I'll use the HI-TECH PICC-18 C compiler in conjunction with MPLAB and a Microchip Technology REAL ICE as my W5100 firmware brewing tools.

About 18 hours later, I returned to write this sentence. I could not get my W5100 to communicate correctly with the PIC18LF8722 to save my life. A cursory look at the W5100 project board didn't indicate any problems. So, I turned to my C code to see if I could find the bug. As it turns out, my C was fine, but my eyes deceived me. A great number of the PIC18LF8722 W5100 address and data I/O pins simply did not get soldered to the W5100 PCB. I use an industrial hot air reflow machine to mount fine-pitched ICs like the W5100 on a regular basis. I've done so many of them that I take the process for granted. Well, this time the reflow machine bit me.

In the meantime, I managed to put some W5100 I/O code together. The official factory W5100 driver code I have is written for AVR devices. So, rather than build my own PIC W5100 include file, I de-Atmeled the factory-supplied W5100 include file. Right now, all I really want from the W5100 include file is the definition code that lays out all of the W5100's internal register addresses. The W5100 factory include file also contains definitions of all of the W5100 register contents, which I'm sure will come in handy later. I lost a bunch of time chasing my soldering snafu, but I gained some of that time back by Microchip-izing the original AVR include file.

My first official W5100 firmware act was to punch the W5100 into a hardware reset (see Listing 1). Since I went to all of the trouble to fix those W5100 address and data solder joints, I'm going to run in W5100 Direct Bus Interface mode. Running in Direct Bus Interface mode means that I don't have to touch the W5100 Mode register, which happens to be the very first W5100 Common register. So, we can run our initial W5100/PIC18LF8722 I/O test on the set of Gateway Address registers at address range 0x0001:0x0004. The Gateway Address register addresses GAR0:GAR3 are defined in the include file I converted, which I renamed w5100_pic.h. As you can see in Listing 1, I put together some basic PIC18LF8722 I/O routines to read and write the W5100 registers. Then, I wrote the contents of the gwayipaddrc array to the W5100's Gateway Address Common register set. To make sure I performed the Common register write, I turned around and read the contents of `GAR0:GAR3` into an array

Listing 1—You won't find this level of coding in the W5100 datasheet examples. Nothing will whizz about without these base register I/O routines.

```
char gwayipaddrc[4] = {192,168,0,1};
char svrmacaddrc[6];

#define make8(var,offset) ((unsigned int)var >> (offset * 8)) & 0x00FF
#define TO_WIZ         TRISF = 0x00
#define FROM_WIZ       TRISF = 0xFF

****************************************
void wr_wiz_addr(unsigned int addr)
{
       addr_hi = (make8(addr,1));
       addr_lo = addr & 0x00FF;
}
void wr_wiz_reg(char reg_data,unsigned int reg_addr)
{
       TO_WIZ;
       wr_wiz_addr(reg_addr);
       data_out = reg_data;
       clr_WR;
       NOP();
       set_WR;
       FROM_WIZ;
}
char rd_wiz_reg(unsigned int reg_addr)
{
       char data;
       wr_wiz_addr(reg_addr);
       clr_RD;
       NOP();
       data = data_in;
       set_RD;
       return(data);
}
****************************************
clr_RSET;
msecs_timer2 = 0;
while(msecs_timer2 < 2);
set_RSET;
addri = GAR0;
for(i8=0;i8<4;++i8)
  wr_wz_reg(gwayipaddrc[i8],addri++);
addri = GAR0;
for(i8=0;i8<4;++i8)
  svrmacaddrc[i8] = rd_wz_reg(addri++);
```

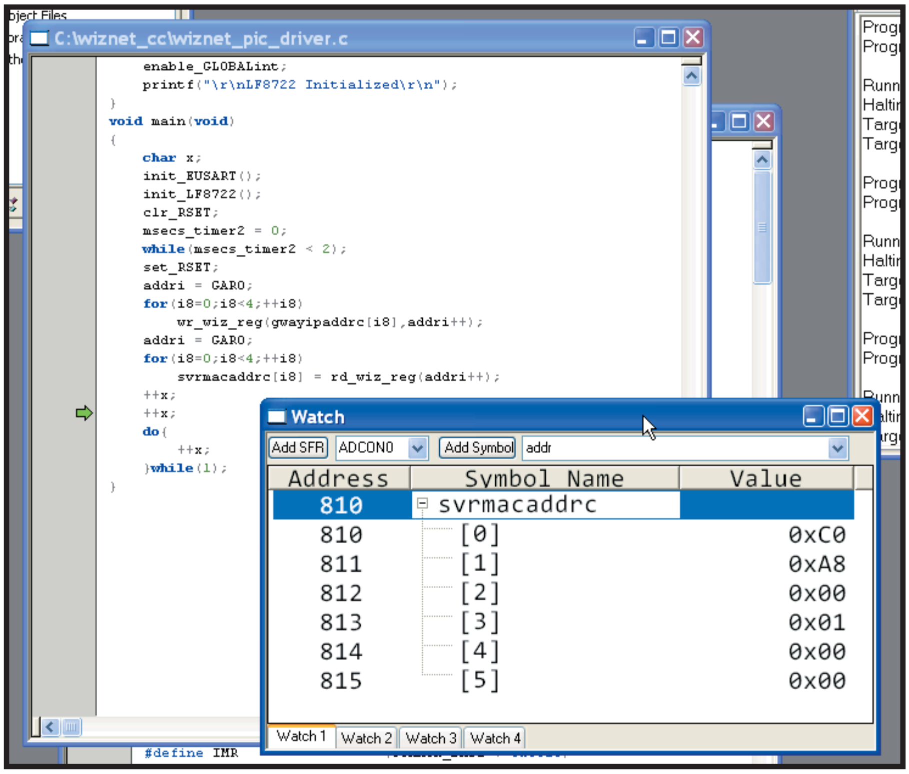
called svrmacaddrc. You can imagine how pleased I was to see the gateway IP address represented in hexadecimal format in the MPLAB WATCH window shot you see in Photo 3. I tested a bit further by using my W5100 register read routines to read the RTR0 Common register pair, which defaults to 0x07D0 and the RCR Common register that follows and defaults to 0x08. All went well. So, I initialized the W5100's gateway, MAC address, subnet mask, and IP address Common registers.

The next step on our way to putting the W5100 project board online involves setting up and defining the socket memory information. We'll use the default of 2-KB-per-socket sizing, which means we don't touch the RMSR (RX memory size) and TMSR (TX memory size) default register values (0x55). As you can see in Listing 2, all we are really doing is establishing the receive and transmit memory boundaries for each of the four sockets that the W5100 supports. With the socket memory allocation task behind us, we can concentrate on what it takes to manipulate a W5100 socket.

The topmost portion of Listing 3 is the code we will execute to open a W5100 UDP socket. The first order of business is to tell the W5100 what type of socket we want to work with. We are working with UDP at the moment. So, I loaded the socket 0 Mode register with a UDP socket value. I already have an application (EDTP Internet test panel) that will send ASCII characters to well-known port 7 and, as you can see in Listing 3, I've loaded the socket 0 Source Port register with 0x0007. We've already loaded our IP and MAC information. Thus, the addition of the UDP source port value enables us to open a UDP socket. From the proliferation of zeros in the Listing 3 socket initialization code, it should be obvious that we will open the W5100's socket 0 in UDP mode.

Once the socket comes online, we have the power to send and receive UDP datagrams. There are a couple of ways to sense an incoming UDP datagram. We can poll the socket's Received Size register or look for the RECV bit in the socket's Interrupt register. As you can see in the UDP datagram receive code that occupies the center section of Listing 3, I have chosen to use the latter.

An incoming UDP datagram sets the RECV bit of the socket's Interrupt register. Our first reaction to this is to clear the RECV bit by writing a "1" to correspond to the RECV bit's position within the Interrupt register. The W5100 takes care of checksums internally and we, as programmers, never see them in our UDP datagram information. The size of the incoming UDP datagram is automatically posted in the socket's Receive Size register. Here, we retrieve the contents of the Receive Size register and place the value into the `get_size` variable. I used the W5100's datasheet variable names where possible to make it a bit easier for you to compare my W5100 driver code with the UDP pseudocode flow example in the W5100 datasheet. The receive buffer's read pointer value is kept in the



```
    enable_GLOBALint;
    printf("\r\nLF8722 Initialized\r\n");
}
void main(void)
{
    char x;
    init_EUSART();
    init_LF8722();
    clr_RSET;
    msecs_timer2 = 0;
    while(msecs_timer2 < 2);
    set_RSET;
    addri = GAR0;
    for(i8=0;i8<4;++i8)
        wr_wiz_reg(gwayipaddrc[i8],addri++);
    addri = GAR0;
    for(i8=0;i8<4;++i8)
        svrmacaddrc[i8] = rd_wiz_reg(addri++);
    ++x;
    ++x;
    do{
        ++x;
    }while(1);
}
```

```
Address   Symbol Name      Value
  810     ⊟ svrmacaddrc
  810       [0]             0xC0
  811       [1]             0xA8
  812       [2]             0x00
  813       [3]             0x01
  814       [4]             0x00
  815       [5]             0x00
```

Photo 3—With the success of reading back what I stored in the WIZnet gateway IP address register, we've established a base of operations for reading and writing the W5100's internal registers.

Listing 2—The W5100 datasheet talks about this with pseudocode. Here's my translation.

```
#define chip_base_address         0x0000
#define RX_memory_base_address    0x6000
#define gS0_RX_BASE               chip_base_address + RX_memory_base_address
#define gS0_RX_MASK               0x0800 - 1
#define gS1_RX_BASE               gS0_RX_BASE + (gS0_RX_MASK + 1)
#define gS1_RX_MASK               0x0800 - 1
#define gS2_RX_BASE               gS1_RX_BASE + (gS1_RX_MASK + 1)
#define gS2_RX_MASK               0x0800 - 1
#define gS3_RX_BASE               gS2_RX_BASE + (gS2_RX_MASK + 1)
#define gS3_RX_MASK               0x0800 - 1
#define TX_memory_base_address    0x4000
#define gS0_TX_BASE               chip_base_address + RX_memory_base_address
#define gS0_TX_MASK               0x0800 - 1
#define gS1_TX_BASE               gS0_TX_BASE + (gS0_TX_MASK + 1)
#define gS1_TX_MASK               0x0800 - 1
#define gS2_TX_BASE               gS1_TX_BASE + (gS1_TX_MASK + 1)
#define gS2_TX_MASK               0x0800 - 1
#define gS3_TX_BASE               gS2_TX_BASE + (gS2_TX_MASK + 1)
#define gS3_TX_MASK               0x0800 - 1
```

socket's Read Pointer register. We will use the read pointer value to form the basis for the variable `get_offset`, whose value we will combine with the socket's receive buffer base address to calculate the beginning address of the UDP datagram's header. The UDP datagram header offered by the W5100 is made up of 4 bytes of destination IP address, 2 bytes of destination port address, and 2 bytes of data size information. Thus, the `header_size` variable value is eight. Once all of the addressing calculations have been made, we can use our W5100 `read register` routine to store the data away in the PIC18LF8722's SRAM for later.

Logically, what is not header information must be data information because we are protected from checksums by the W5100 architecture. With that, we can deduce that the `udp_data_size` variable will contain the number of data bytes we need to retrieve and store. Again, using our home-brewed W5100 I/O code, we read the data from the W5100 receive buffer memory and store it in the appropriate PIC18LF8722 SRAM locations. Our absorption of the UDP datagram and its header is complete. We end our receive session by issuing the RECV command in the socket's Command register, which updates the receive buffer pointers. For those of you who are following along with the pseudocode flow in the W5100 datasheet, note that the `udp_data_size` variable is not a W5100 datasheet variable. It's a Fred variable.

Sending a UDP datagram is very similar to receiving one. We'll reuse the information we received earlier and bounce a UDP message back at the sender. Recall that our received UDP datagram header contained a destination IP address and a destination UDP port value. We thought ahead and stored both of the header values. Now all we have to do is retrieve them from the PIC18LF8722's SRAM and load them into the proper W5100 Socket registers. I begin my UDP datagram transmission in that manner within the bottom portion of the code in Listing 3. Using the socket's transmit buffer write pointer, I calculate where in the W5100 transmit buffer to begin stuffing the data I wish

Listing 3—Think about it. All you ever do with any communications device is receive and transmit. I pulled the logic behind this code from the pseudocode flow in the W5100's datasheet.

```
//SOCKET INTI******************************************************
do{
wr_wiz_reg(Sn_MR_UDP,Sn_MR(0));      //protocol = UDP
wr_wiz_reg(0x00,Sn_PORT0(0));        //well-known ECHO port
wr_wiz_reg(0x07,Sn_PORT1(0));
wr_wiz_reg(Sn_CR_OPEN,Sn_CR(0));     //give the open command
if(rd_wiz_reg(Sn_SR(0)) != SOCK_UDP) //wait for the socket to come online
    wr_wiz_reg(Sn_CR_CLOSE,Sn_CR(0));
}while(rd_wiz_reg(Sn_SR(0)) != SOCK_UDP);

//RECEIVE**********************************************************
do{
//look for incoming UDP datagrams
i16 = rd_wiz_reg(Sn_IR(0));
}while(i16 == 0);
wr_wiz_reg(0x04,Sn_IR(0));
//get the datagram size
hi_byte = rd_wiz_reg(Sn_RX_RSR0(0));
lo_byte = rd_wiz_reg(Sn_RX_RSR1(0));
get_size = make16(hi_byte,lo_byte);
//get the datagram's buffer offset
hi_byte = rd_wiz_reg(Sn_RX_RD0(0));
lo_byte = rd_wiz_reg(Sn_RX_RD1(0));
get_offset = make16(hi_byte,lo_byte) & gS0_RX_MASK;
//calculate the datagram's starting buffer address
get_start_address = gS0_RX_BASE + get_offset;
//UDP header size
header_size = 8;
//store the UDP header information
addri = get_start_address;
for(i8=0;i8<header_size;++i8)
{
    packet[ip_destaddr+i8] = rd_wiz_reg(addri++);
    ++get_offset;
}
//store the UDP data
get_start_address = gS0_RX_BASE + get_offset;
udp_data_size = get_size - header_size;
addri = get_start_address;
for(i8=0;i8<udp_data_size;++i8)
{
    packet[UDP_data+i8] = rd_wiz_reg(addri++);
    ++get_offset;
}
//update the receive buffer pointers
wr_wiz_reg(Sn_CR_RECV,Sn_CR(0));

//TRANSMIT*********************************************************
//load destination IP address
addri = Sn_DIPR0(0);
for(i8=0;i8<4;++i8)
    wr_wiz_reg(packet[ip_destaddr+i8],addri++);
//load destination port address
addri = Sn_DPORT0(0);
for(i8=0;i8<2;++i8)
    wr_wiz_reg(packet[UDP_srcport+i8],addri++);
//get transmit buffer offset
hi_byte = rd_wiz_reg(Sn_TX_WR0(0));
lo_byte = rd_wiz_reg(Sn_TX_WR1(0));
get_offset = make16(hi_byte,lo_byte) & gS0_TX_MASK;
//calculate transmit data buffer start address
get_start_address = gS0_TX_BASE + get_offset;
//load data into transmit buffer
addri = get_start_address;
for(i8=0;i8<udp_data_size;++i8)
{
    wr_wiz_reg(packet[UDP_data+i8],addri++);
    ++get_offset;
}
//update transmit buffer pointer
wr_wiz_reg((make8(get_offset,1)),Sn_TX_WR0(0));
wr_wiz_reg((make8(get_offset,0)),Sn_TX_WR1(0));
//send data
wr_wiz_reg(Sn_CR_SEND,Sn_CR(0));
while(rd_wiz_reg(Sn_CR(0)));
```

to transmit. I then transfer the previously stored UDP datagram data from the PIC18LF8722's SRAM into the W5100's transmit buffer. The W5100 will transmit the data located between the socket's transmit read pointer and transmit write pointer. So, I must update the transmit write pointer by increasing it by the number of bytes I need to transmit. Once that's done, I issue the SEND command and wait for the send success signal, which is a cleared socket Command register. I can now issue a CLOSE command in the socket's Command register to close the socket or send or receive another UDP datagram.

CONGRATULATIONS!

You have completed W5100 bootcamp. In addition to the W5100 register I/O code, UDP transmit code, and UDP receive code, you have a basic and flexible W5100 hardware design to work with.

Here's a hint that will help you determine very early on where your W5100 design stands: Execute only the code through loading the IP address. Load the gateway address, the MAC address, the subnet mask, and the IP address. Don't open any sockets. At this point, you can PING your W5100 design. If you get good PING returns, your PHY hardware and your W5100 register I/O code are good to go. You've also tested and confirmed the operation of your W5100 address, data, and control signals. Bringing up UDP is a fun and easy way to get to know the W5100. The EDTP Internet test panel is a UDP application that runs on your PC. The EDTP Internet test panel is available for download from www.edtp.com. ■

Fred Eady (fred@edtp.com) has more than 20 years of experience as a systems engineer. He has worked with computers and communication systems large and small, simple and complex. His forte is embedded-systems design and communications.

PROJECT FILES

To download code, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2007/208.

SOURCES

EDTP Internet test panel
EDTP Electronics, Inc.
www.edtp.com

HI-TECH PICC-18 C Compiler
HI-TECH Software
www.htsoft.com

PIC18LF8722 Microcontroller, REAL ICE, and MPLAB
Microchip Technology, Inc.
www.microchip.com

RDI-125BAG1A Pulse transformer
U.D. Electronic Corp.
www.ude-corp.com

W5100 TCP/IP Ethernet controller
WIZnet, Inc.
www.ewiznet.com